The Use of Object-Oriented Design Patterns in the SAMRAI Structured AMR Framework[*]

Richard D. Hornung[†] Scott R. Kohn[†]

Abstract

We describe the use of object-oriented design patterns in the implementation of a flexible framework for structured adaptive mesh refinement. We present five common patterns—*Smart Pointers*, *Singleton*, *Abstract Factory*, *Strategy*, and *Chain of Responsibility*—that have greatly simplified software development. These design patterns have enabled the decomposition of complex algorithms into smaller, more manageable, decoupled components that may be reused across a variety of applications.

1 Introduction

The design and implementation of quality, high-performance numerical software frameworks is difficult. Framework designers must address issues of algorithm complexity, evolving research requirements, and software reuse within the targeted application domain. Modern numerical algorithms, such as structured adaptive mesh refinement methods [3, 4], consist of many complex numerical components involving sophisticated time integration methods, various geometry descriptions, time interpolation, spatial refinement and coarsening, and linear and nonlinear solvers. These numerical components interact in complex ways that must be captured in the design of the software infrastructure. Finally, numerical frameworks are usually developed in tandem with research projects in algorithms and applications; thus, the framework software must be designed to evolve as computational scientists improve their understanding of application domains and the associated numerical methods.

In this paper, we address some of these design issues in the context of a parallel structured adaptive mesh refinement (SAMR) framework called SAMRAI. Object-oriented techniques and design patterns [8] have been valuable tools for the high-level organization of the SAMRAI software architecture. They have enabled us to isolate various functional parts of complex algorithms into different framework components so that applications can be built from smaller algorithmic "building blocks." As a result, we provide a flexible software library that simplifies the management of inherently complex SAMR algorithms and is being applied to diverse SAMR applications.

This paper is organized as follows. We begin with a brief overview of SAMRAI and the basic SAMR methodology. Section 3 describes five different design patterns—*Smart Pointers* [7], *Singleton*, *Abstract Factory*, *Strategy*, and *Chain of Responsibility* [8]—used in

[*]This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract number W-7405-Eng-48. Published in the *Proceedings of the SIAM Workshop on Object-Oriented Methods for Interoperable Scientific and Engineering Computing*, Yorktown Heights, NY, October 21-23, 1998. LLNL report UCRL-JC-131825.

[†]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551, `http://www.llnl.gov/CASC/SAMRAI`, `hornung@llnl.gov`, `skohn@llnl.gov`.

the SAMRAI framework. Finally, Section 4 discusses the usefulness of design patterns and object-oriented techniques within the SAMRAI framework and the general computational science community.

2 The SAMRAI Framework

Structured adaptive mesh refinement has shown great potential as a numerical simulation methodology for a variety of applications in computational fluid dynamics [1, 2, 12], laser-plasma interactions [6], radiation transport [11], porous media [10], and materials [9, 13, 14]. However, SAMR methods are not widely used in the scientific computing community. The primary reason for this is that SAMR codes are complex and require a substantial amount of software infrastructure to support productive application development. Fortunately, many software components are common across diverse problem domains and may be incorporated into a general-purpose infrastructure that supports a broad range of applications.

SAMRAI is a C++ object-oriented framework that provides computational scientists with general and extensible software support for prototyping and developing parallel SAMR applications. The primary goal of the SAMRAI effort is to explore the use of SAMR technology in new problem domains and to develop new numerical and algorithmic approaches for more traditional SAMR applications. SAMRAI provides an overarching software architecture that orchestrates the various processes involved in a complex numerical simulation. SAMR applications can usually be decomposed into smaller, simpler constituent parts such as algorithmic components, data structures, and numerical routines. In the process of building an application with SAMRAI, computational scientists select the appropriate numerical and algorithm components from the framework and supply only those operations that are specific to their application. Thus, the computational scientist leverages a large simulation code base and only specializes certain components as needed.

A full description of SAMR algorithms is well beyond the scope of this paper. However, we provide a brief overview of the basic SAMR approach to help in understanding the algorithmic and software issues in the remainder of this paper. The SAMR approach, introduced by Berger, Oliger, and Colella [3, 4], represents simulation data using a hierarchy of nested levels of spatial and temporal mesh refinement. The hierarchy dynamically adapts to follow interesting features in the evolving simulation and focuses computer resources in these localized regions of the computational domain.

A SAMR hierarchy consists of several mesh levels. All computational cells at a particular level in the hierarchy represent the same mesh resolution. Each level consists of a collection of *patches*, each of which is a logically rectangular collection of computational cells. A patch contains data that represent simulation quantities in the region of the simulation domain covered by the patch region. The level with the coarsest mesh resolution defines an abstract, global integer index space. Then, each successively finer level is a refinement of a portion of the next coarser index space. The organization of the computational mesh into a hierarchy of levels of patches allows data communication and computation to be expressed as geometrically-simple, efficient operations. Consequently, the SAMR methodology is used to construct an application code from a set of computational tasks, each of which is expressed in terms of operations on mesh patches.

In the remainder of this paper, we discuss object-oriented techniques used to implement two of SAMRAI's design goals. First, SAMRAI must support a wide range of complex data structures on SAMR patches, including arbitrary user-defined types. Second, SAMRAI must provide flexible and extensible algorithm support for a variety of SAMR applications.

One important design constraint is that SAMRAI must enable new application development and support new user-defined data types without forcing changes to the underlying framework source code or recompilation of the libraries.

3 Design Patterns in SAMRAI

Design patterns are specific solutions to common, recurring software engineering problems. Each pattern codifies a general solution technique by providing a problem description, the solution pattern, and a list of consequences resulting from the application of the pattern. In practical terms, a design pattern describes a configuration of a small set of objects whose cooperative behavior solves a software design problem. There are several useful books that describe design patterns [5, 8, 15]; our discussion follows that of Gamma *et al.* [8] most closely.

We have found design patterns to be very helpful in solving some key design problems that arose during the construction of the SAMRAI software architecture. Some of these patterns are covered in detail in the following five sections. We begin each section with a discussion of a design problem encountered in the SAMRAI framework. We then describe the design pattern selected to solve that particular design problem and the consequences of that decision.

Section 3.1 describes the *Smart Pointer* pattern which simplifies the management of dynamically allocated memory and provides safe dynamic type casting. The *Singleton* pattern (Section 3.2) defines a single point of contact for objects shared among various components. The *Abstract Factory* creational pattern (Section 3.3) enables SAMRAI to support new user-defined patch data types without requiring modifications to the framework. Finally, the *Strategy* (Section 3.4) and *Chain of Responsibility* (Section 3.5) behavioral patterns are used in SAMRAI to decouple various framework components and thus obtain greater reuse of fundamental algorithm pieces.

3.1 Smart Pointers

In this section, we describe two typical problems in the SAMRAI framework that are solved through the use of *Smart Pointer* [7] techniques: (1) safe dynamic type casting and (2) memory management for shared objects. The need for safe dynamic type casting is illustrated by the following example. As described in Section 3.3, all SAMRAI patch data types share a common base class called `PatchData`:

```
class PatchData : public ... {
   void copy(const PatchData& source) = 0;
   ...
};
```

The concrete data types that are instantiated on SAMRAI patches—such as `CellData<double>` or `EmbeddedBoundaryData`—inherit the signature for `copy()` declared in `PatchData`. However, concrete classes often require the type of the `copy()` argument to be the same as the class itself, not any arbitrary `PatchData` object. For example, it would probably not make sense to copy `EmbeddedBoundaryData` into `CellData<double>`. Unfortunately, there is no way to enforce this through the C++ type system at compile-time. Although templates are often used in similar cases to ensure type safety, they are not sufficient for complex applications that must access data through abstract base classes.

Run-time type safety can be achieved through the use of run-time dynamic type casting. In this case, dynamic type casting of the argument `source` within the `copy()`

implementation returns a pointer to the object if the cast is valid and returns NULL otherwise. Although dynamic type-safe casting is part of the `C++` standard, it is not yet supported by all `C++` compilers.

Another common problem solved through *Smart Pointers* is memory management for shared objects. In this case, many framework objects maintain pointers to a shared object instance that must be deallocated when all references to it disappear. Since ownership of this shared object cannot be uniquely established, the application cannot easily determine when to deallocate it. For example, SAMRAI patches typically share a pointer to a patch descriptor object. Moreover, patches are created and destroyed dynamically during mesh refinement. The memory allocation problem is solved with reference counting smart pointers that track the number of references to an object and then delete the pointed-to object when the number of references decrements to zero.

3.1.1 Pattern Description The *Smart Pointer* pattern is a common `C++` pattern [7]. It consists of two parts: a templated `Pointer` class that manages the object reference counting and a collection of classes that support run-time safe type casting. All pointed-to objects are required to inherit from a common base class and provide a small number of functions to implement the type conversions.

3.1.2 Consequences The use of the *Smart Pointer* pattern within SAMRAI has greatly simplified the management of dynamic memory allocation; multiple objects may share pointers to the same object and the smart pointers guarantee that there will be no memory leaks. The type-safe dynamic casting ensures that type errors will be caught at run-time.

The primary disadvantage of the *Smart Pointer* approach is that it introduces a common base class for all pointed-to classes. While not a burden when writing new code, it is esthetically unappealing to force otherwise unrelated classes to inherit from a common base class, since it introduces extraneous coupling in the software architecture.

3.2 Singleton Classes

Some classes in the SAMRAI framework are intended to be instantiated only once, with that single instance shared by various entities. For example, a `VariableDatabase` object contains information about the variables used in a computational simulation (e.g., pressure, density, or velocity). The database must be accessible to all algorithm components to extract information about the variables and their roles in the simulation. Traditionally, such shared objects were implemented using global variables; however, global variables do not ensure only one instance of a class and they do not allow extension by subclassing. Instead, we implement shared objects such as `VariableDatabase` using the *Singleton* creational pattern as described in Gamma *et al.* [8]

3.2.1 Pattern Description The *Singleton* pattern ensures that a class will have only one instance and provides global, well-defined access to that instance. Also, the class may be extended through inheritance. Then, clients may use the subclass object without changes to their code.

In SAMRAI, the `VariableDatabase` encapsulates its single instance and maintains strict controls over access to this instance. It declares a `getDatabase()` static member function that returns a pointer to the single database instance. In addition, the constructor and destructor of the class are protected to ensure that only the database and its subclasses may create an instance of the database.

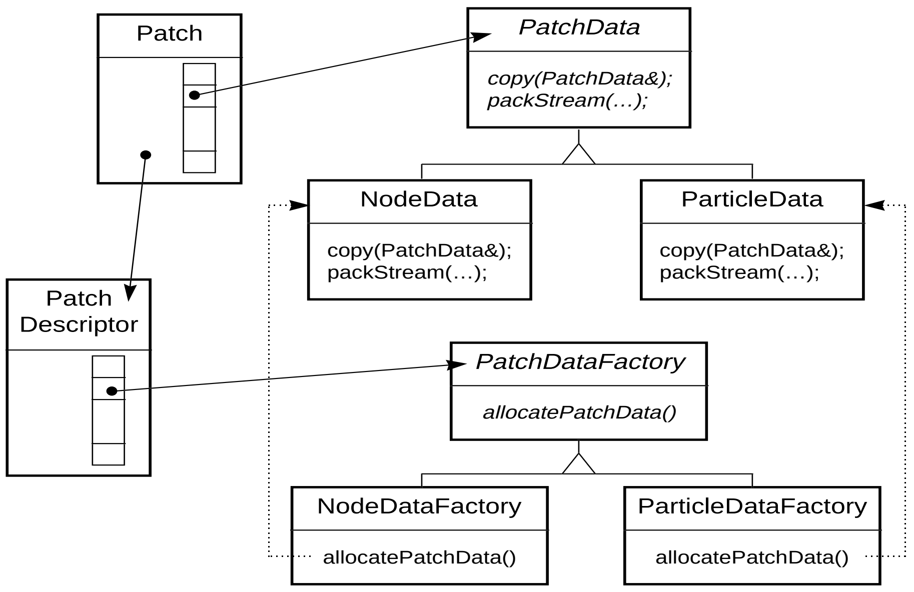

FIG. 1. *The* Abstract Factory *pattern manages the allocation of data for the SAMRAI patch hierarchy. As illustrated by the dotted lines, subclasses of* `PatchData` *are created by associated subclasses of* `PatchDataFactory`. *This diagram follows the OMT (Object Modeling Technique) notation [8].*

3.2.2 Consequences The *Singleton* pattern provides a more flexible alternative to the use of global variables. The name space remains cleaner and applications may use extensions of a basic singleton object, even at run-time. A singleton can be extended through standard class derivation and any client can use the subclass without needing changes in its own code.

3.3 Abstract Factory

Recall that one of the primary considerations in the design of SAMRAI was the need to support complex user-defined data on an SAMR patch hierarchy. Patches in an SAMR application may contain data such as cell-centered arrays of doubles, node-centered arrays of integers, or user-defined collections of particles among other types. Patch data types are manipulated by the framework, which manages allocation, deallocation, copying, and marshaling and unmarshaling of data for communication between processors.

We believe that SAMRAI users should not modify the framework software or recompile the libraries to add new data types, as such practices violate sound software engineering principles. Thus, the framework cannot know the concrete class types for user-defined patch data, since these classes may be designed and implemented long after the framework has been compiled. In this case, how can the framework allocate user-defined data? Clearly, SAMRAI cannot execute `new` for concrete types that do not exist at compile-time. The solution to this problem is the *Abstract Factory* creational pattern.

3.3.1 Pattern Description The *Abstract Factory* pattern describes an approach for creating families of related objects without specifying their concrete classes [8]. This pattern does so through two related inheritance hierarchies. The first hierarchy is rooted in an *abstract product* class that declares the interface for all objects created by the pattern. These product objects are created by *factory* objects in a second hierarchy.

Figure 1 shows how this pattern is implemented in the SAMRAI framework. The SAMRAI `Patch` is a container class for all patch data types that exist in some rectangular region of index space. All patch data types inherit from an abstract `PatchData` class and define a set of required routines such as `copy()` and `packStream()` (used for interprocessor communication). Each `Patch` has a smart pointer to a `PatchDescriptor` that contains the factory objects needed to make the concrete patch data. Then, to create an instance of a `PatchData` object, the `Patch` consults the `PatchDescriptor` and asks the appropriate `PatchDataFactory` to allocate a `PatchData` instance. The `allocatePatchData()` function in the factory returns the concrete `PatchData` instance.

3.3.2 Consequences The *Abstract Factory* pattern separates concrete object creation and declaration by encapsulating the responsibility for creating product objects. This pattern enhances software flexibility and extensibility since concrete product classes (such as `NodeData` in Figure 1) never appear in the framework code. Thus, new product classes can be added after the framework has been compiled and archived into a library.

There are two drawbacks to *Abstract Factory* pattern. First, every new product class requires the definition of two new classes—the product class and the factory class. Second, some form of dynamic safe type casting is needed to obtain concrete class references. For example, although it is sufficient for the `Patch` container class to manipulate data as abstract `PatchData` objects, user-defined numerical routines will need to extract data from the patch and process that data using the concrete class interface. The cast from abstract product to concrete product requires some form of run-time type checking such as that described in Section 3.1.

3.4 The Strategy Pattern

SAMR applications involve sophisticated procedures that can be decomposed into smaller constituent parts. These parts include algorithms for sequentially advancing a set of SAMR patch levels, integrating single patch levels, dynamically changing the mesh, and numerical routines defined on individual patches. A primary goal of SAMRAI is to provide a flexible algorithmic framework that encapsulates components such as these so that they may be reused in different SAMR applications when appropriate.

Developing a flexible algorithmic framework is difficult. The most important research challenge is discovering how complex algorithms may be factored into their constituent parts. Then, the specific behavior of each component must be determined and appropriate interfaces must be defined between the different pieces. Ideally, each individual algorithmic part may be replaced or enhanced without adversely influencing the behavior of the other components. If this separation is attained, it becomes relatively easy to combine existing software components to construct a complete SAMR algorithm. While we are still grappling with these issues in the development of SAMRAI, we believe that the approach outlined here demonstrates substantial progress.

The *Strategy* design pattern is the primary object-oriented design technique that we use to encapsulate algorithmic units and define reusable interfaces between software components. Next, we illustrate our use of this pattern by describing the decomposition of a standard SAMR algorithm into its primary components.

3.4.1 Pattern Description The intent of the *Strategy* pattern is to define and encapsulate families of algorithmic components to make them interchangeable through

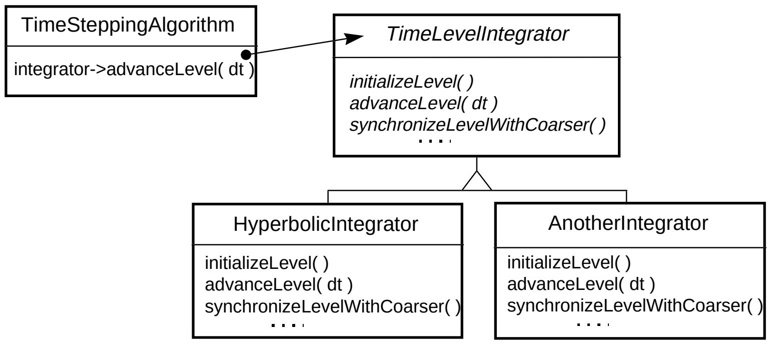

FIG. 2. *SAMRAI uses* Strategy *to define a family of time-dependent integration algorithms.*

common interfaces. Consequently, this pattern is well-suited to our concerns. A concise example of the basic form of the *Strategy* pattern is illustrated in Figure 2.

In SAMRAI, a `TimeSteppingAlgorithm` class controls a sequence of timesteps that advances the levels in an SAMR hierarchy. While this class is fairly general, the routines that advance data on the individual levels are specific to each application. When a `TimeSteppingAlgorithm` object is created, it is configured with a suitable level integration algorithm object. The level integration class may be supplied by the framework; for example, the `HyperbolicIntegrator` class is provided for systems of hyperbolic conservation laws. Otherwise, another integrator class must be implemented (e.g., `AnotherIntegrator`). Each level integrator class is derived from the `TimeLevelIntegrator` abstract base class and must satisfy the interface defined by that class. The `TimeSteppingAlgorithm` object maintains a pointer to the abstract type; thus, it knows nothing of any specific, concrete level integration process.

Figure 3 shows multiple *Strategy* patterns, including a particular instance of the pattern in Figure 2, combined to form a complex algorithm from simpler components. The configuration represents a common SAMR algorithm for treating hydrodynamics applications, such as the Euler equations of gas dynamics, with explicit timestepping [3, 4].

At the top algorithmic level, the `TimeSteppingAlgorithm` class controls the overall SAMR scheme. It is configured with `HyperbolicIntegrator` and `RichardsonExtrapolation` objects, which supply routines to advance the data and dynamically adjust the mesh, respectively. Consistent with the *Strategy* pattern, the timestepping algorithm knows only the abstract types `TimeLevelIntegrator` and `GriddingAlgorithm`.

The *Strategy* pattern is repeated in the design of `RichardsonExtrapolation` and `HyperbolicIntegrator`. Concrete subclasses of `MeshGenerator` and `LoadBalancer` (not shown) provide routines that create box regions and load balance the patches on a new patch level. The `EulerPatchModel` class supplies numerical routines for the Euler equations on a single patch in the mesh hierarchy. Both `HyperbolicIntegrator` and `RichardsonExtrapolation` invoke functions in `EulerPatchModel` (e.g., numerical flux computation, conservative difference, select cells for refinement, etc.), but they are independent of the specific routines. That is, the `HyperbolicIntegrator` has a pointer to `HyperbolicPatchModel` and `RichardsonExtrapolation` has a pointer to `RichExtrapPatchModel`. The `EulerPatchModel` class, derived from both of these abstract base classes, implements functions declared in both of their interfaces.

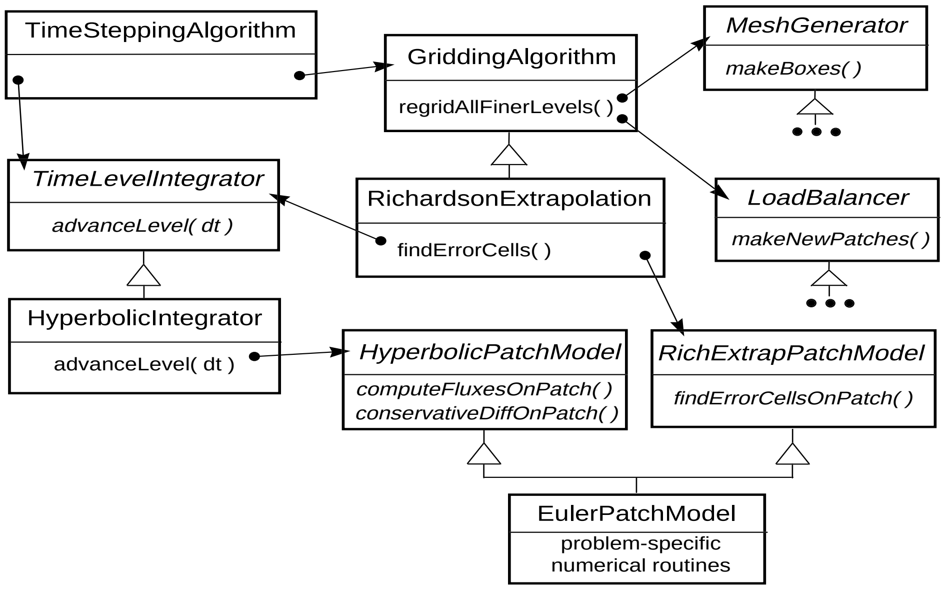

Fig. 3. *Multiple instances of the* Strategy *pattern are combined in SAMRAI to build a complex AMR solution from simpler components.*

3.4.2 Consequences The *Strategy* pattern provides a useful degree of algorithmic encapsulation in SAMRAI. Using common interfaces to characterize families of related algorithmic components, a system may be configured to perform a wide range of behaviors. This type of "plug-and-play" interoperability is advantageous for several reasons. First, it frees application programmers from unnecessary, redundant code implementation and reduces development time. Second, it promotes the exploration of different algorithmic choices within a single application. Third, it increases software reuse within the framework, which facilitates testing, maintenance, and extensibility of the architecture.

The encapsulation forced by the *Strategy* pattern is a valuable alternative to large, overly-complex classes that can occur through the abuse of inheritance. For instance, the design in Figure 2 could have been implemented by inheriting from `TimeSteppingAlgorithm` directly. The result would be several larger, more complicated classes that differ in level integration procedures, but have much timestepping code in common. Although decoupling the algorithm components slightly increases function call overheads, the cost is negligible at the high algorithmic level.

3.5 Chain of Responsibility

Data motion between SAMR hierarchy patches requires time interpolation, coarsening, and refinement operators that depend on problem geometry, the type of patch data, and the centering of patch data. The SAMRAI parallel communication routines are defined in terms of abstract operator and geometry base classes to decouple them from the details of the particular geometry or concrete operators used in an application. The association between a patch data type and its concrete operators is managed through the SAMRAI geometry classes, which are responsible for cataloging the operators for a particular patch data type.

As users define new patch data types for their applications, they must also provide the required operators for these types. However, the geometry classes cannot know the concrete types of these new operators, since they were defined after the compilation of the SAMRAI framework. Thus, the geometry classes require an extensible lookup mechanism that allows the definition of new operators for user-defined patch data types. This particular design problem is solved by the *Chain of Responsibility* design pattern.

3.5.1 Pattern Description The *Chain of Responsibility* pattern avoids coupling the sender of a request to any potential request receiver by giving multiple object handlers an opportunity to handle the request. Our implementation of this pattern follows Gamma [8].

To obtain operators, an algorithm object queries a geometry object for the operators associated with various patch data types. The geometry object passes each request to the chain of handlers it owns. The request is forwarded along the chain until the correct operator handler is found. This handler then returns a pointer to the desired operator. The correct operator handler is found when the patch data type of the request matches the patch data type of the handler, where the type equality is determined using the dynamic casting facilities described in Section 3.1.

3.5.2 Consequences There are several advantages to using the *Chain of Responsibility* pattern for the operator lookup. First, this pattern reduces the coupling between patch data types, operators, and the algorithms that use them since these objects have no explicit knowledge of each other's concrete types. The same mechanism may be used for arbitrary patch data types and operators without changing any of the algorithm code. Second, the system is sufficiently flexible so that new concrete operator handlers (thus, new operators) may be added to the chain at run time. In particular, there is no need to use conditional statements or enumerated types that cannot be extended without recompilation. Third, the use of the dynamic cast mechanism ensures type safety. That is, an operator cannot be associated with a patch data type if the patch data type is not of the type supported by the operator handler.

A disadvantage of the *Chain of Responsibility* pattern is that it requires the implementation of an operator class and hander class for each concrete operation. The potentially-large number of classes may be reduced by bundling operators together within larger classes and using conditionals to choose the correct behavior. However, the overall amount of source code required in either case is about the same. In most applications, each chain is traversed only once for each variable. Once an operator is found and a pointer to its instance is returned, the operator may be called directly through the pointer. No future use of the chain is required. We believe that the general flexibility that we achieve using the chain mechanism far outweighs the drawbacks for our framework.

4 Summary and Conclusions

Object-oriented design patterns have been very useful in the design and development of the SAMRAI structured adaptive mesh refinement software architecture. By using patterns such as *Abstract Factory*, *Strategy*, and *Chain of Responsibility*, we have simplified the management of complex SAMR algorithms. Consequently, design patterns have enabled us meet two of our most important design goals: flexible, extensible algorithm support for a wide range of SAMR applications, and generic support for arbitrary patch data types.

When considering the adoption of object-oriented techniques, the scientific computing community has often focused on implementation and performance issues associated with "low-level" classes such as vectors, matrices, arrays, and `C++` STL containers. While these abstractions are useful, we feel that object-oriented design offers the most benefit at the higher levels of a numerical software architecture. Object-oriented techniques enable the decomposition of complex algorithms into smaller, more manageable pieces that are suitable for a variety of applications. They promote code and algorithm reuse and also facilitate testing and management of software framework components. Most importantly, object-oriented patterns support more productive application construction by allowing rapid exploration of new algorithms that are built from both existing and new components.

References

[1] M. Aftosmis, M. Berger, and J. Melton, *Adaptation and surface modeling for cartesian mesh methods*, in Proceedings of the 12th AIAA Computational Fluid Dynamics Conference, San Diego, CA, June, 1995, 1995. AIAA Paper 95-1725.

[2] A. S. Almgren, J. B. Bell, P. Colella, L. H. Howell, and M. L. Welcome, *A conservative adaptive projection method for the variable density incompressible navier-stokes equations*, Tech. Rep. LBNL-39075, Lawrence Berkeley National Laboratory, Berkeley, CA, 1996.

[3] M. J. Berger and P. Colella, *Local adaptive mesh refinement for shock hydrodynamics*, Journal of Computational Physics, 82 (1989), pp. 64–84.

[4] M. J. Berger and J. Oliger, *Adaptive mesh refinement for hyperbolic partial differential equations*, Journal of Computational Physics, 53 (1984), pp. 484–512.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *A System of Patterns*, John Wiley and Sons, New York, NY, 1996.

[6] P. Colella, M. Dorr, and D. Wake, *Numerical simulation of plasma fluid equations using locally refined grids*, Tech. Rep. UCRL-JC-129913, Lawrence Livermore National Laboratory, Livermore, CA, 1998. submitted to J. Comp. Phys.

[7] J. Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley Publishing Co., Menlo Park, CA, 1992.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Menlo Park, CA, 1995.

[9] F. X. Garaizar and J. A. Trangenstein, *Adaptive mesh refinement and front tracking for shear bands in an antiplane shear model.* to appear in *SIAM Journal on Scientific Computing*, 1998.

[10] R. D. Hornung and J. A. Trangenstein, *Adaptive mesh refinement and multilevel iteration for flow in porous media*, Journal of Computational Physics, 136 (1997), pp. 522–545.

[11] J. P. Jessee, L. H. Howell, W. A. Fieveland, P. Colella, and R. B. Pember, *An adaptive mesh refinement algorithm for the discrete ordinates method*, in Proceedings of the 1996 National Heat Transfer Conference, Houston, TX, August 3-6, 1996, 1996.

[12] R. I. Klein, J. B. Bell, R. B. Pember, and T. Kelleher, *Three dimensional hydrodynamic calculations with adaptive mesh refinement of the evolution of rayleigh taylor and richtmyer meshkov instabilities in converging geometry: Multi-mode perturbations*, in Proceedings of the 4th International Workshop on Physics of Compressible Turbulent Mixing, 1993.

[13] S. Kohn, J. Weare, E. Ong, and S. Baden, *Software abstractions and computational issues in parallel structured adaptive mesh methods for electronic structure calculations*, in Proceedings of the Workshop on Structured Adaptive Mesh Refinement Grid Methods, Minneapolis, MN, March 1997, Springer-Verlag.

[14] J. A. Trangenstein, *Adaptive mesh refinement for wave propagation in nonlinear solids*, SIAM J. Sci. Stat. Comput., 16 (1995), pp. 819–839.

[15] J. Vlissides, *Pattern Hatching: Design Patterns Applied*, Addison-Wesley Publishing Co., Menlo Park, CA, 1998. Software Patterns Series.